This Page Is Inserted by IFW Operations
and is not a part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images, please do not report the images to the Image Problems Mailbox.

THIS PAGE BLANK (USPTO)

THIS PAGE BLANK (USPTO)

(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) Publication number : **0 524 719 A2**

(12) EUROPEAN PATENT APPLICATION

(21) Application number : 92304936.5

(22) Date of filing : 29.05.92

(51) Int. Cl.^{5} : **G06F 9/44**

(30) Priority : 29.05.91 US 707121

(43) Date of publication of application :
27.01.93 Bulletin 93/04

(84) Designated Contracting States :
DE FR GB IT

(71) Applicant : DELL USA L.P.
9505 Arboretum Boulevard
Austin, Texas 78759 (US)

(72) Inventor : Chan, Wai-Ming Richard
1100 215 Research, Apt. 2045
Austin, Texas 78759 (US)
Inventor : Schieve, Eric W.
10513 Scotland Well
Austin, Texas 78750 (US)
Inventor : Zeller, Charles P.
1819 Travis Heights Boullevard
Austin, Texas 78704 (US)
Inventor : Abbott, Gary W.
6302 Jennings Drive
Austin, Texas 78727 (US)

(74) Representative : Dowler, Angus Michael et al
Abel & Imray Northumberland House 303-306
High Holborn
London, WC1V 7LH (GB)

(54) **Computer system with alterable bootstrapping software.**

(57) A computer which carries its BIOS in a Flash EPROM. A UV-EPROM carries a redundant BIOS, which can be overlaid onto the BIOS address space by selection with a physical switch.

The BIOS contains a small core software program, at the BIOS entry point, which checks BIOS integrity, and provides for reloading the Flash EPROMS's BIOS if needed (from a floppy disk, or by copying the entire contents of the UV-EPROM).

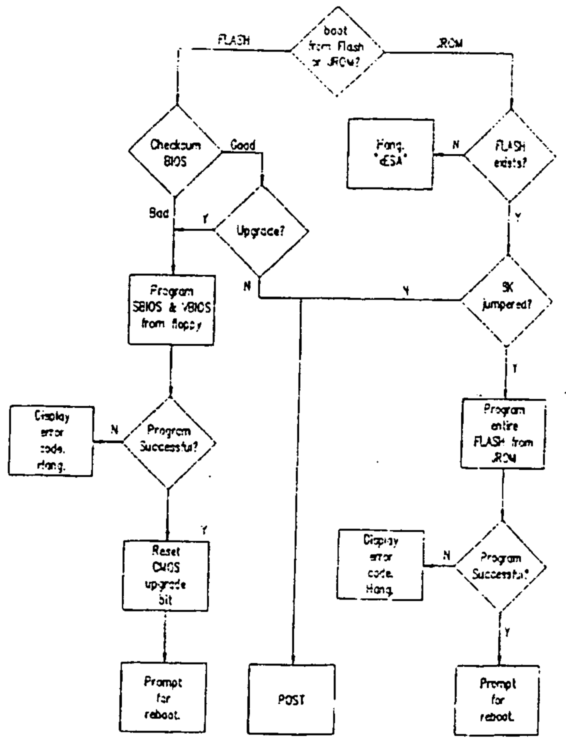

Figure 1

EP 0 524 719 A2

The present invention relates to a computer system.

Any computer system must have some way to begin program execution after a cold start. The hardware architecture of a CPU (processor) will normally provide for a "reset" operation, which places all of the hardware circuits in a known electrical state; but it is still necessary to start the CPU on execution of a desired program. For example, in the very early days of computing, some computer systems would be manually configured to read in a "bootstrap loader" program at startup. This bootstrap program was a simple program which loaded in, and started execution of, another sequence of instructions, which were the beginning of the desired program. Bootstrap programs are often referred to simply as "boot" software.

To give a more recent example, the Intel 80x86 microprocessors, after a hardware reset will always attempt to begin program execution from a specific memory address. That is, the microprocessor will read the contents of that memory location, and will attempt to execute the bits it finds there as a microprocessor instruction. Thus, if a branch (or conditional branch) instruction is found at this address, the microprocessor will continue its program execution from whatever address' is specified. The specific memory location used by the 80x86 family is xxxFFFFOh, i.e. 16 bits below the top of the memory space. Other microprocessors may use a different starting address, but similar principles apply.

Thus, this initial target adress is the entry point for every session of use. This address is normally used to enter execution of programs which must be run every time the computer is used.

Whatever hardware is used will have its own procedures to return to a known state when a reset occurs. However, at some point these procedures end, and the CPU is ready to begin execution of instructions.

At this point the system performs various overhead tasks under software control. These may include surveying the system configuration, sanity checks etc.

In modern personal computers, the initial target address is normally used as the entry point to a "basic input/output system" (BIOS) program. The BIOS program contains frequently-used routines for interfacing to key peripherals, for interrupt handling, and so forth.

Thus, the BIOS software provides some degree of machine-independence. However, in PC-class computers, this independence is not fully exploited by the available commercial software. Many programs bypass the BIOS software, and directly access the underlying hardware addresses or devices. See Glass, "The IBM PC BIOS", Byte, April 1989, pp. 303ff. In addition to these basic input/output routine's, the "BIOS" software also includes some key pieces of overhead software, such as configuration update and the power-on-self-test (POST) routines.

The POST routines provide an extensive check for hardware integrity. The BIOS software will also launch the machine into the operating system software.

Depending on how the system has been set up, the BIOS software may direct program execution into DOS, Unix, PS/2, a DOS variant, or another operating system. However, the choice of operating system is not particularly relevant to the invention described in the present application. (Thus, the term "BIOS" has become somewhat broader nowadays, and normally refers to this whole collection of basic system routines.)

If the BIOS software were to become corrupted, the computer could become unusable. Thus, the BIOS software has conventionally been stored in read-only memory (ROM). When the microprocessor attempts to access the initial target address, it reads out software from the BIOS ROM.

In 1980 there was only one source for IBM-compatible BIOS software, and that was from IBM. However, during the 1980s, as IBM-compatible personal computers became more popular, modified versions of IBM-compatible BIOS ROMs were developed, and IBM compatible BIOS ROMs were offered by multiple vendors. As of 1991, BIOS software is often modified to implement system-dependent features, especially in low-power systems.

Improvements in BIOS software mean that sometimes it will be desirable to implement a BIOS upgrade. Dedicated users have successfully pried out and replaced ROM chips, but most users would not want this degree of hands-on contact.

Some attempts have been made in the past to provide capability for updating the basic system software. See e.g., Bingham, D.B., "Achieving flexible firmware," 1978 MIDCON Technical Papers at 20/3/1-4(1978).

It is believed that some vendors may have offererd upgradable-BIOS systems. However, insofar as is known, no system has offered protection against corruption while updating BIOS.

Without boot software, a computer could not run any program at all. Thus, corruption of the boot software can make a computer totally unusable. Thus, for security against such corruption, personal computers have commonly been manufactured with their BIOS software in nonvolatile memory chips. However, for rapid system development and upgrading, it is desirable to be able to rapidly provide new BIOS chips, or even to upgrade existing BIOS chips. Over years, a variety of developments in semiconductor device technology have been used to reconcile these needs.

One of the simplest nonvolatile memories is the mask ROM. By custom-patterning one level of a chip, a complex pattern of data can be permanently encoded. For example, an array with polysilicon row lines accessed by row decoders, and metal column lines accessed by column decoders, can have contact holes selectively etched, so that one bit of data is encoded at each row/column intersection. Alternatively, polysilicon row lines can be selectively linked to diffused column lines by metal shorting straps. With this technology, a new mask must be prepared whenever any bit of data is changed. In high volume, mask ROMs are reasonably cheap, but they are not suited for rapid upgrading.

A "programmable read-only-memory," or PROM, can be electrically written. PROMs are typically built using solid-state fuses (or antifuses), which will become open (or shorted) whenever a high current is driven through them. Thus, the programming of a conventional PROM requires the application of a high voltage (e.g. 10 or 15 V) to achieve the high currents needed. However, once the data has been written it is permanent.

An "electrically programmable read-only memory," or EPROM, can be electrically written, and can be erased by ultraviolet light. EPROMs are typically built using a floating-gate avalanche MOS device. This is a MOS transistor with an additional isolated thin film (the floating gate) interposed between the MOS control gate and the channel. By pumping charge into the floating gate, the effective threshold voltage of the MOS transistor can be changed, and this threshold voltage change is detected by a sense amplifier. To write a cell, the channel is typically driven hard, with a high voltage on the control gate. Secondary carrier multiplication in the channel creates hot electrons, which are attracted into the floating gate. Thus the programming of a conventional EPROM requires the application of a high voltage (e.g. 10 or 15 V) to achieve the high currents needed. Moreover, programming is typically quite slow, e.g. several milliseconds per bit. To erase a cell, it is exposed to ultraviolet light. The energetic photons excite energetic carriers, which can pass through the dielectric layer to neutralize the charge on the floating gate. Thus EPROMs normally need a package with a quartz window. EPROMs are very commonly used, since they are cheap and their timing standards are familiar. EPROMs are also referred to as UVPROMs.

An "electrically erasable programmable read-only-memory," or EEPROM or E^{2}PROM, can be electrically written and electrically erased. EEPROMs, like EPROMS, are typically floating-gate devices. However, in an EEPROM the write and erase operations typically use tunnelling. Thus, the programming of a conventional EEPROM requires the application of a high voltage (e.g. 15 or 20 V). Programming and erasure are typically quite slow, e.g. several milliseconds per bit. EEPROMS are less commonly used than EPROMs, since they tend to be more expensive and to require even higher voltages.

A more recent modification of the EPROM is the "Flash EPROM." This device, like the EEPROM is electrically erasable, but only in blocks. Although this device does not have the bit-by-bit programmability of the EEPROM, it is still useful in many applications.

A wide variety of rewritable nonvolatile memory technologies have been proposed, and doubtless others will continue to be proposed. For example, the computers of the 1950s and early 1960s used "core" memory technology, which is, to some extent, nonvolatile and rewritable.

The disclosed computer system contains features which permit the boot software to be easily upgraded and protect against corruption of the boot software. The disclosed computer system may use Flash EPROMs or other memory technologies

The disclosed system uses a rewriteable nonvolatile memory as the primary boot memory. For further security against corruption of the boot software, the preferred system uses two boot memories. By selection with a hardware switch, either one can be connected as the boot source.

Since either of the two boot memories may need to be the target, both boot memories are mappable to the same address. Note that this causes some difficulty in writing to the volatile boot memory: if the volatile boot memory is not in the memory map, it cannot be written to.

In the presently preferred embodiment, this difficulty is avoided by mapping the core software out to RAM, and then commanding an address configuration change to access the other boot memory. If one boot memory is being copied onto the other, then the contents of the source memory must also be copied out.

As noted above, the default mapping of the boot memory address, at initial power-up, is controlled by a physical switch. However, thereafter, a peripheral memory controller chip controls the mapping of the boot memory address. Thus, by issuing a command to this peripheral chip, the core software can change this mapping.

Thus, to write to the boot memory, the core software copies itself out to a RAM location, and commands the microprocessor to branch into the RAM address. The bit to select the boot memory is then toggled, so that writes to the boot memory addresses are now directed to the other boot memory (i.e. to the boot memory which was not the source of the code being executed). The desired data can then be written into the target boot memory. Note that, if the source of the data is in the active boot memory, that code must be written out into RAM before the boot memory address toggle is switched.

The starting address in the boot software is occupied by a small (and strongly protected) software core, which performs sanity checks, and also provides the needed supervisory functions for boot software upgrading and replacement.

Thus, the disclosed computer system solves the problem of performing BIOS restoration/upgrades in-situ (i.e. without removing the FLASH boot memory ROM from the system). The end-user (or a service technician) can program BIOS code into the FLASH without any specialized extenal hardware or software ROM-programming tools.

After any hard reset of the system, the core software detects whether the boot software is corrupted, and whether the user has entered a request for upgrade. If these conditions have occurred, the core software performs the FLASH programming from the appropriate data source.

In the presently preferred embodiment, the core software provides for the following three classes of cases. These cases together provide coverage for all conceivable field conditions:

First case:

While executing reset code from the FLASH, the system BIOS and/or video BIOS are found to have a bad checksum. In this case, power-on-self-test and booting is impossible, therefore, the boot code prompts the user to insert a diskette containing system and video BIOS code and reprograms the FLASH.

In the presently preferred embodiment, this is done with the assistance of a small 4-character diagnostic display which is mounted directly on the system chassis. This display is referred to as SmartVu™. When the system and video BIOS programming is successfully completed, the user is prompted (via SmartVu) to reset the system. Normal POST and boot should follow.

Second case:

While executing reset code from the FLASH, the boot code detects a "CMOS" flag setting indicating a user requested upgrade. (Personal. computers normally contain a battery-backed CMOS memory which stores configuration parameters; in system discussions, this memory is often referred to merely as the "CMOS" although non-CMOS memories can be used). The boot code then prompts the user (via SmartVu) to insert a diskette containing the upgrade system and video BIOS code. When the system and video BIOS programming is successfully completed, the user is prompted (via SmartVu) to reset the system. Normal POST and boot should follow.

Third case;

While executing reset code from the UVPROM, the boot code detects that a FLASH is present and that its protected 8k boot sector is jumpered for programming. Since the UVPROM is intended as a secondary ROM device, this condition is understood as a request for FLASH programming and the entire contents of the UV-PROM, including the boot code in the upper 8k sector is copied into the FLASH. When the entire FLASH is successfully programmed, the user is prompted to reset the system. If the FLASH is selected as the primary ROM device, normal POST and boot should occur from the FLASH.

Note that the foregoing steps may require the CPU to ascertain, while it is running boot code, which memory is the source of the code. This is accomplished, in the presently preferred embodiment, by letting the CPU read a register in a peripheral chip.

The preferred rewritable boot memory is a sector-protected Flash-EPROM. The sector which contains the software core is actually protected by a jumper, which must be physically moved before the software core can be overwritten. This provides additional robustness.

Because the core software may have to read from floppy disk, it includes the necessary overhead routines to perform this. These routines are generally similar to the BIOS functions under INT13h.

The invention provides a computer system including a rewritable non-volatile memory which holds bootstrapping instructions including instructions capable of effecting the over-writing of instructions held in the rewritable non-volatile memory, from an external source, following a hard reset of the system.

The invention also provides a computer system including a rewritable non-volatile memory and a second non-volatile memory which holds bootstrapping instructions including instructions capable of effecting the over-writing of instructions held in the rewritable non-volatile memory from an external source, and means for selecting either the second non-volatile memory or the rewritable non-volatile memory as its source of bootstrapping instructions, following a hard reset of the system.

The first non-voltaile memory may be a mask ROM, PROM or UVPROM which holds the basic system operating instructions including the bootstrap and BIOS instructions required for system activation. Those instructions may be substantially unalterable, and the system can be activated or reactivated by placing it under the control of the first non-volatile memory

Preferably, the bootstrapping instructions held in the second non-volatile memory are such as to effect the over-writing of instructions held in the rewritable non-volatile memory by the bootstrapping instructions held in the second non-volatile memory, following a hard reset of the system.

The rewritable non-volatile memory may be a FLASH EPROM or an electrically programmable and erasable ROM to which can be written instructions, including instructions which effect the writing of instructions from an external source to the rewritable non-volatile memory. Thus, instructions can be written to the rewritable non-volatile memory and the rewritable non-volatile memory can then be used as the source of activating instructions for the computer system, with the flexibility that the computer system can itself rewrite at least some of the instructions held in the rewritable non-volatile memory. The computer system can be returned to the operating configuration provided by the instructions held in the first non-volatile memory at any time by means of a user-operated switch, say.

Preferably, the computer system includes manually operable means which has a first setting for preventing the over-writing of instructions held in the rewritable non-volatile memory and a second setting permitting the over-writing of instructions held in the rewritable nonvolatile memory.

The provision of a means which is only operable manually to inhibit the over-writing of selected instructions held in the rewritable non-volatile memory makes it possible to protect those selected instructions from accidental alteration.

Preferably, the instructions are such as to make the computer system capable of writing the instructions from the second non-volatile memory to the rewritable non-volatile memory so that the instructions held in the second non-volatile memory can be readily written to the rewritable non-volatile memory.

Preferably, the computer system includes a user settable switch for selecting the second non-volatile memory as its source of bootstrapping instructions and is capable of reversing the switch setting to select the rewritable non-volatile memory as its source of activating instructions thereafter. That is, the computer system is capable of activation in accordance with the instructions held in the second non-volatile memory, following which it can write those instructions to the rewritable non-volatile memory, after which it switches control to the rewritable non-volatile memory.

Preferably, the computer system includes multi-element diagnostic display means capable of indicating the result of an operation to over-write instructions held in the rewritable non-volatile memory.

The diagnostic display means may be a multi-character display, for example, a 4-character display linked to a sub-system for checking on whether the writing of system and video BIOS has been successful and providing an apparopriate indication.

The first non-volatile memory may be rewritable and include means capable of preventing its data from being overwritten. That is, the first non-volatile memory may, in practice, be a rewritable non-volatile memory with substantial protection against its data being overwritten.

The invention provides a method of operating a computer system including the steps of writing bootstrapping instructions from a second non-volatile memory to a rewritable non-volatile memory following a hard reset of the system and, thereafter, reading the bootstrapping instructions from the rewritable nonvolatile memory in bootstrapping the computer system.

Preferably, the method of operating the computer system includes the steps of reading initial bootstrapping instructions from the rewritable nonvolatile memory in bootstrapping the computer system and overwriting the remaining instructions in the rewritable non-volatile memory with data from an external source.

Preferably, the method of operating the computer system includes the steps of reading bootstrapping instructions from the rewritable non-volatile memory, subjecting the bootstrapping instructions to validity checks as they are read and, when a validity check is failed, overwriting the bootstrapping instructions held in the rewritable non-volatile memory with the bootstrapping instructions held in the second nonvolatile memory.

The method for operating the computer system may comprise the steps, whenever the system has undergone a hard reset, of immediately performing the following operations in a programmable central processing unit (CPU):

if the said CPU is executing software from a rewritable nonvolatile boot memory,

performing a checksum operation on the basic system software in the said rewritable nonvolatile boot memory, and, if a checksum error is found,

prompting the user to provide a data source for the basic system software, and thereafter,

reprogramming the said rewritable nonvolatile boot memory from the data source provided by the user.

A modified method of operating the computer system comprises:

reading at least one bit from a predetermined data location, and if the said bit is in a first state, prompting the user to provide a data source for the basic system software, and thereafter,

reprogramming the said rewritabole nonvolatile boot memory from the data source provided by the user.

If the said CPU is executing software from a second nonvolatile boot memory which is not the same as the said rewritable nonvolatile boot memory and if the said rewritable nonvolatile memory is not currently write-protected, THEN the contents of the said rewritable nonvolatile memory are overwritten with the contents of the second non-volatile boot memory.

The method of operating the computer system may comprise the steps, whenever the system undergoes a power-up transition, of immediately performing the following operations in the system microprocessor:

ascertaining whether the said microprocessor is currently executing boot software from a rewritable nonvolatile boot memory or from a second nonvolatile boot memory which is not the same as the said rewritable boot memory but which is mappable onto the same address as the said rewritable boot memory,

if the said microprocessor is executing software from said rewritable boot memory,

performing a checksum operation on the basic system software in the said rewritable boot memory, and if a checksum error is found,

prompting the user to provide a data source for the basic system software, and thereafter,

reprogramming the said rewritable boot memory from the data source provided by the user, followed by prompting the user to reboot the system.

The method of operation at a power-up transition may include reading at least one bit from a predetermined data location and if the said bit is in a first state, prompting the user to provide a data source for the basic system software, and thereafter,

reprogramming the said rewritable boot memory from the data source provided by the user followed by prompting the user to reboot the system.

If the said bit is not in the first state, the system executes a power-on-self-test routine from the said rewritable boot memory.

If the said microprocessor is executing software from said second boot memory and if said rewritable nonvolatile memory is not currently write-protected the system copies the entire contents of the said second boot memory and overwrites the contents of the said rewritable nonvolatile boot memory, thereafter prompting the user to reboot the system.

If any other condition exists, the system executes a power-on-self-test routine from the said second boot memory.

The method for operating the computer system may comprise, during normal operation, the steps of:

executing a sequence of instructions in at least one CPU which can fetch instructions in a programmable sequence from a memory;

initiating a system reset operation which includes a step of resetting the said CPU when a power restoration or software reset command occurs,

executing in the said CPU, substantially immediately after the said step of resetting the said CPU, a core software program, stored in a rewritable nonvolatile boot memory, which tests the integrity of data in the said rewritable boot memory, and prompts the user to supply replacement data for said rewritable boot memory is corrupt.

If the said CPU executes the said core software program successfully, then a power-on-self-test program is executed subsequently in the said CPU.

If the said CPU executes the said power-on-self-test software program successfully, then the said CPU is launched on the execution of operating system software.

Preferably, the said rewritable nonvolatile boot memory consists essentially of a Flash EPROM or an electrically programmable and erasible read-only-memory.

Preferably, the said second nonvolatile boot memory consists essentially of a programmable-read-only-memory which is not electrically erasable.

The present invention will be described with reference to the accompanying drawings, which show embodiments of the invention wherein:

Figure 1 is a flow chart which shows key of the invention.

Fig. 2 shows a hardware configuration which permits switching between alternative boot memories in an embodiment of the invention.

The preferred embodiment is an 80486-based EISA-bus PC system. General features of the EISA bus are described by Glass, "Inside EISA," Byte magazine November 1989, pp/ 417ff, and in the EISA specification. General features of the Intel 80486 are described by Sartore, "The 80486: A Hardware Perspective," Byte Magazine IBM Special Edition, Fall 1989, pp. 67ff. Further detailed background on the 80486 may be found in the "80486 Programmer's Reference" and the "80486 Hardware Reference Manual," both available from Intel.

Many of the architectural features of this system are conventional in modern PC systems. However, several unusual features are used. One of these is the provision for dual boot memories, with hardware and software switching betwen them. Another notable feature is the use of the "SLOB" controller chip described below.

In the presently preferred embodiment, the system motherboard includes a custom chip, referred to herein as the "SLOB" chip, which performs a variety of special functions. These include reset control, X-bus transceiver control, and control of the boot memories and CMOS nonvoltaile memory.

The "X-bus," or "extension bus," is an extension of the system bus, but is not directly connected to it. Instead, data is selectably transferred from the S-bus to the system bus, or vice versa, by bidirectional transceivers. The X-bus is commonly used, in PC architectures, to provide easier loading requirements for a variety of devices on the motherboard.

The software core, in the presently preferred embodiment, resides in the protected 8k boot sector of a FLASH ROM or in the upper 8k of a 128k UVPROM. Selection of the primary ROM device is accomplished via a physical switch.

Figure 2 shows more detail of how this is implemented. The SLOB chip supplies chip enable (CE) signals separately to the FLASH EPROM 220 and to the UVEPROM 230 (and also to the NVRAM 240). An output enable line OE is connected to both boot ROMs, and address lines are also provided to both. A hardware switch provides a line ROMIN to the SLOB chip, to define which boot memory is the initial default target.

In the presently preferred embodiment, the FLASH EPROM has sector-by-sector protection. (This feature is found in the 28F001B type chip available from Intel).

In the presently preferred embodiment, one sector 220A, of only 8K bytes, is dedicated to the core software. This sector is protected by a hardware jumper, so corruption of the core software is unlikely.

To achieve automatic programming of the Flash memory from a plug-in ROM, the hardware must provide a switching mechanism that allows software to boot from one source and then toggle between sources. This switchability is also required to access EISA configuration from Flash if running BIOS out of a plug-in ROM.

A user settable hardware switch determines which boot memory supplies the boot code at power-up or cold boot. This switch sets the polarity of a bit called ROMIN which is decoded by the hardware control logic to enable the selected boot source. This bit is readable from and I/O port, so that software can determine the source of the boot code, i.e. ROM(bit=0) or Flash (bit=1). The software then has the ability to toggle subsequently between accessing ROM or Flash by setting a bit that assumes the opposite state of ROMIN after a cold boot. This bit is called ROMEN and enables Flash when 0 and ROM when 1. This bit is readable as well as writable via an I/O port. This mechanism allows hardware to boot from ROM then gives software the ability to determine the presence of Flash memory and program Flash automatically by transferring the contents of the BIOS in ROM to the Flash.

The chip referred to herein as the TRANE chip is a custom memory controller. Besides performing normal DRAM management functions, this chip also provides selection, by memory domains, of which memory areas will be cached or not.

The actual implementation of the core software will now be described in detail by way of example only. It must be understood that this specific implementation is merely illustrative.

Figure 1 is a flow chart which schematically shows key portions of the methods used in the computer system of the presently preferred embodiment.

When the 80486 first comes out of reset, it comes up in real mode (but the high address bits are held high, so that the processor can go to the top of the 4-Gigabyte (32-bit) memory space. The processor accesses address FFFFFFF0h. See Glass, "Protected Mode", Byte Magazine December 1989, pp. 377ff.

Preferred Assembly Language Implementation

The actual implementation of key portions of the core software, in the presently preferred embodiment, will now be given. The following listings in documented assembly language also contain a large number of informal comments. These comments do not necessarily define the scope of the invention, but will help to explain the motivation, structure, and working of the presently preferred embodiment.

Some of the procedures actually used, in the presently preferred embodiment, will now be described in detail. Of course, it should be understood by those skilled in the art that the very specific implementation details given are not by any means necessary to the invention. The following wealth of detail is provided merely to assure compliance with the best mode requirements of U.S. patent laws.

Procedure 8KBOOT_STRT

Procedure 8kboot_strtis the power-on-reset entry point into the 8k boot sector code for a 128k FLASH EPROM. This code along with a reset vector is located in the upper 8k boot sector of a 128k FLASH EPROM. The reset vector will be located at 1fff0h (physical ROM address), and the start of this routine will be located at 1e000h (start of the 8k boot sector). The reset vector will contain a NEAR jmp to this code, so execution will start in processor real mode, ROM native mode, at logical address 0ffffe000h. This procedure implements most of the steps in the flow chart of Figure 1. Specifically, this procedure:

- Saves the state of EAX and DX for later use by POST.

- Initializes a base address for the DRAM controller (the TRANE chip, in the presently preferred embodiment).
- Disable interrupts (NMI and INTR).
- If warm boot, jmp into POST SHUTDOWN routine.
- Else, if power-on check to see if we're running out of a UVPROM or the FLASH.
- If FLASH, jump to the "bootFLASH" procedure which programs the flash if the BIOS is corrupted or if the user has requested an upgrade.
- If UVPROM, jump to the "bootUVPROM" code which copies the UVPROM to the FLASH if a FLASH exists.
- If either "bootFLASH" or "bootUVPROM" does not program the flash, control returns to "do_reboot" which displays a SmartVu message and halts. NOTE: Because of the redundancy between this code and early parts of POST, (RESET, SHUTDOWN) the BIOS is NOT dependent on this code. This means that a working BIOS can be built for a 64k non-FLASH ROM architecture by simply removing this module from the build process. The original 64k BIOS reset/initialization logic has been left untouched.

```
; ---------------------------------------------------------------
CODE8K      SEGMENT USE16 PUBLIC 'CODE'
            ASSUME cs:CODE8K, ds:CODE8K

            extrn   int13h  : near
            extrn   initFd  : near
            extrn   dskprm  : byte
```

```
fdcDOR          equ 3F2h      ;Digital output register.
                        ;  bit7 = 0 Reserved.
                        ;  bit6 = 0 Reserved.
                        ;  bit5 = 1 Enable drive 1 motor.
                        ;  bit4 = 1 Enable drive 0 motor.
                        ;  bit3 = 0 Enable floppy interrupts and DMA.
                        ;  bit2 = 0 Controller reset.
                        ;  bit1 = 0 Reserved.
                        ;  bit0 = 0 Select drive 0.
                        ;       = 1 Select drive 1.

        PUBLIC @8kboot_strt

@8kboot_strt:
; Save ax in high word of ebp, dx in high word of esp.
        mov bp,ax
        shl ebp,16
        mov sp,dx
        shl esp,16

; Disable primary and secondary caches. 486 cache comes up enabled so we want
; it OFF as fast as possible.
            DIS_486CACHE

; Grab a base IO address for TRANE.  The first IO write address after
; power-on will be appropriated by TRANE for it's base address.
        mov al,0            ;Valid TRANE index for warm boot case.
        out TRANE_BASE,al   ;Dummy IO write to set TRANE base.

; Flush secondary cache.
        mov dx,slob_portXX
        in  al,dx
        or  al,slob_portXX_flushBit     ;Hold ext. cache in flush.
        out dx,al
        and al,NOT slob_portXX_flushBit ;Reset ext. cache flush.
        out dx,al

; Turn off interrupts and NMI.
        cli
        mov al,NMIOFF+00h
        out NMIMSK,al         ;Turn off NMI's
        WAFORIO
        in  al,NMIMSK+1

; CPU in fastest possible mode
        mov al,PITSL2+PITRLL+PITMDI
        out XPITMD,al

        cld

; Warm or cold boot? If cold boot, we do the 8k code. If warm boot we
; bail into SBIOS reset logic.
        IN  AL,UPISTA        ;see if 8042 system bit on,
        TEST    AL,04H          ;bit 2 of 8042 status port
        jz  SHORT is_cold_boot   ;If warm boot, haul ass!

; shutdown code = CMOS shutdown byte
        MOV AL,NMIOFF+CMSHUT
        OUT CMOSAD,AL
        WAFORIO
        IN  AL,CMOSDT        ;get CMOS shutdown byte
        MOV AH,AL

; This is a CPU reset, not a bus reset. In this case we may assume RAM
; is initialized and therefore RSTFLG can be used to determine the sense
; of shutdown 0s. If it's a shutdown 0 and RSTFLG == 1234h, then it's a
; warm boot, else it's a shutdown that wants to act like a cold boot.
        cmp ah,0             ;Shutdown 0?
        jne is_warm_boot        ;If not, warm boot.
        mov bx,ROMDAT
        mov ds,bx
        ASSUME  DS:ROMDAT
        cmp RSTFLG,1234h        ;CTRL-ALT-DEL ?
        ASSUME  DS:CODE8K
        je  SHORT is_warm_boot   ;If so, warm boot.

; Switch to protected mode and do a FAR jmp to the 4G native mode BIOS
; to remain in the 8k boot code.
is_cold_boot:
        mov sp,OFFSET pm_retptr
        jmp pmode            ;Protected mode entry routine.
pm_retptr   DW  OFFSET pm_ret
```

```
; If we're booting from FLASH, look for a corrupted BIOS or a user
; requested upgrade.
; If we're booting from UVPROM, program the FLASH if it exists.
pm_ret:     mov dx,slob_extRom       ;SLOB external ROM register.
            in  al,dx
            test    al,slob_extRom_romIn     ;are we booting from UVPROM or FLASH?
            jz  bootUVPROM       ;booting from UVPROM.
            jmp bootFLASH        ;booting from FLASH.

            jc  do_reboot        ;Display error message.
; Return here from "bootUVPROM" or "bootFLASH" if FLASH programmed
; successfully.  Issue SmartVu message to reboot and wait forever.
do_reboot:  mov esi,eax          ;Save SMARTVU message.

;Make sure the checksum is still good.
            mov di,OFFSET cmosSum_ret2
            jmp cmosSum          ;Checksums CMOS routine.
cmosSum_ret2:

;Beep and display error message.
            mov cx,800h          ;Set frequency.
            mov di,OFFSET beep_ret
            jmp beep
beep_ret:   mov eax,esi          ;Restore SMARTVU message.
            mov cx,0             ;Delay forever.
            jmp flash_msg        ;Display error message.

; Do a FAR jmp to the SBIOS reset logic at F000:E1FF. Restore the reset state
; prior to the jmp.
back2RM:    mov dx,slob_portXX       ;Access portXX.
            in  al,dx            ;Read in current value.
            waforio
            and al,not slob_portXX_flhPrg;Disable FLASH program/erase.
            out dx,al            ;Write out the new value.

            PAR_RESET            ;Reset parity flop at port 61h.

;Make sure the checksum is still good.
            mov di,OFFSET cmosSum_ret
            jmp cmosSum          ;Checksums CMOS routine.
cmosSum_ret:    OPSIZE
```

```
        lidt    FWORD PTR cs:real_idt    ;Interrupts from vector table @ 0.

; Set 64k limits for all selectors.
        mov ax,GDTD_8KBOOT
        mov ds,ax
        mov es,ax
        mov ss,ax
        mov fs,ax
        mov gs,ax

        mov eax,cr0
        and al,NOT 1
        mov cr0,eax           ;Real mode

; Set segments regs back to CPU reset defaults.
        xor ax,ax
        mov ds,ax
        mov es,ax
        mov ss,ax
        mov fs,ax
        mov gs,ax

is_warm_boot:
;Restore ax and dx to their reset values.
        shr esp,16
        shr ebp,16
        mov dx,sp             ;Restore reset value of DX.
        mov ax,bp             ;Restore reset value of AX.

        DB  0eah              ;FAR jmp to SBIOS reset entry point.
        DW  0E1FFh
        DW  0f000h
```

Procedure *BOOTUVPROM*

Procedure *bootUVPROM* contains the 8k power-on logic executed when a UVPROM is installed. When cold-booting from a UVPROM, this code is executed to look for a FLASH ROM and if present, attempts to copy the entire UVPROM contents into the FLASH. If FLASH programming is unsuccessful for any number of reasons, e.g., FLASH is not present, 8k boot sector is not jumpered for erase/programming, etc., then this routine just bails back into POST.

```
; ------------------------------------------------------------

        PUBLIC   bootUVPROM
bootUVPROM   PRCC     NEAR

        mov sp,OFFSET $106_retptr
        jmp $init_vl106
$106_retptr  DW   OFFSET $106_ret
$106_ret:    mov eax,'RAMi'
        mov dx,SMARTVU
        out dx,eax

; Initialize RAM.
        mov bp,OFFSET RAMinit_ret     ;Point sp to near return address.
        jmp RAM_init           ;Initialize Trane.

;Map in the first available SIMM at 0-2Mb.
RAMinit_ret:     mov al,CMSMTPL+NMIOFF
        out CMOSAD,al
        in  al,CMOSDT          ;Get SIMM map from CMOS.
        movzx    bx,al              ;Save SIMM configuration in bx.
        mov dh,al
        mov ax,GDTD_4G         ;4G data seg ...
        mov ds,ax              ;into ds.
        mov es,ax              ;into es.
findSIMM:    bsf bp,bx              ;Find first installed SIMM position.
        jnz SHORT mapSIMM
        mov ax,GDTD_8KBOOT
        mov ss,ax
        jmp JROM2post          ;If none, do POST.
```

```
mapSIMM:    mov dx,bp
        TRANE_CUT_IX    TRANE_RCROM,dl   ;Program 1st SIMM at 0 as a 256kbit part.

; Copy 8k boot from ROM to RAM at 11e000h (1M + 128k - 8k).
        mov eax,' x8K'
        mov dx,SMARTVU
        out dx,eax
        mov ecx,2000h SHR 2       ;8kb -- 2k DWORDs.
        mov edi,11e000h      ;Destination in RAM above 1M.
        mov esi,0ffffe000h       ;Source in 4G ROM.
        cld              ;Forward MOVSD.
        ADSIZE
        rep movsd            ;move 8k boot from 4G ROM to RAM above 1M.

; Check the RAM code against the ROM to verify the copy. (and validate RAM)
        mov eax,'8Kck'
        out dx,eax
        mov ecx,2000h SHR 2       ;8kb -- 2k DWORDs.
        mov edi,11e000h      ;Destination string in RAM above 1M.
        mov esi,0ffffe000h       ;Source string in 4G ROM.
        ADSIZE
        repe    cmpsd            ;Search for non-matching DWORD.
        je  SHORT jmp2RAM        ;If RAM/ROM match then jump to the RAM code.

; If the 8k boot in RAM is corrupted then assume that SIMM is bad and mark
; it out in CMOS. Go back and look for the next good SIMM.
        btr bx,bp            ;Reset the current SIMM bit in the map.
        mov al,CMSMTPL+NMIOFF
        out CMOSAD,al
        mov al,bl
        out CMOSDT,al        ;Disable the SIMM in the CMCS map.
        jmp findSIMM         ;Look for the next good SIMM.

; Jump into the 8k boot code that has been copied to RAM.
jmp2RAM:    JMPP    GDTC_1M_RAM      ;Load cs with selector for 1M RAM code.

;Relocate the GDT.
        OPSIZE
        lgdt    FWORD PTR cs:ram_gdt_ptr

; Set up some stack.
```

```
        mov ax,GDTD_STACK
        mov ss,ax
        mov sp,8000h

; initialize timer and calibrate speed.
        pusha
        push    ds
        call    init_timer
        call    calb_spd
        pop ds
        popa

; Disable the UVPROM and enable the FLASH.
        mov dx,slob_extRom        ;FLASH/UVPROM enable register.
        in  al,dx
        and al,NOT slob_extRom_romEn ;Enable FLASH.
        out dx,al

; Look for signature to determine FLASH presence.
        mov BYTE PTR fs:[flash_CP],flash_signature
        mov al,fs:[flash_SIG_lo]          ;Lo byte of signature.
        cmp al,FLASH_MAN_CODE         ;28F001B manuf. code?
        jne SHORT no_flash            ;If not, FLASH not present, do POST.
        mov al,fs:[flash_SIG_hi]          ;Hi byte of signature.
        cmp al,FLASH_DEV_CODE0        ;28F001B device code?
        je  SHORT uvprom2ram          ;If so, go on.
        cmp al,FLASH_DEV_CODE1        ;Alternate 28F001B device code?
        je  SHORT uvprom2ram              ;If so, flash present.

; Flash not present, so look for an EISA NVRAM.  If none, then hang.
no_flash:   mov dx,slob_configA
        in  al,dx
        test    al,slob_configA_nvRamIn      ;Is an NVRAM installed?
        jnz SHORT no_EISA_cfg         ;If not, hang.
        mov ax,GDTD_1M_RAM            ;Stack segment in current code.
        mov ss,ax
        jmp JROM2post             ;Got NVRAM for EISA, but no FLASH, so do POST.

no_EISA_cfg:     mov eax,'xESA'            ;EISA CMOS not present.
        jmp do_reboot             ;Display error message.
```

```
; FLASH is present.
; Copy and verify the 128k UVPROM into RAM.
uvprom2ram: mov eax,'xBIO'
        mov dx,SMARTVU
        out dx,eax

        mov dx,slob_extRom        ;FLASH/UVPROM enable register.
        in  al,dx
        or  al,slob_extRom_romEn      ;Enable UVPROM
        out dx,al

        mov ecx,18000h SHR 2
        mov edi,100000h       ;Destination in RAM above 1M.
        mov esi,0fffe0000h        ;Source in 4G ROM.
        mov ax,GDTD_4G        ;4G data seg ...
        mov ds,ax             ;into ds.
        mov es,ax
        cld               ;Forward MOVSD.
        ADSIZE
        rep movsd             ;move 128k UVPROM to RAM above 1M.

; Check the RAM code against the ROM to verify the copy. (and validate RAM)
        mov eax,'BIOc'
        mov dx,SMARTVU
        out dx,eax
        mov ecx,18000h SHR 2
        mov edi,100000h       ;Destination string in RAM above 1M.
        mov esi,0fffe0000h        ;Source string in 4G ROM.
        ADSIZE
        repe    cmpsd             ;Search for non-matching DWORD.
        je  SHORT erase_flash     ;If RAM/ROM match then erase all of flash.

; RAM failed during the UVPROM copy.
        JMPP    GDTC_8KBOOT       ;Jump back to UVPROM.
        OPSIZE
        lgdt    FWORD PTR cs:gdt_ptr

; Mark out the current SIMM in CMOS and go back to look for the next
; good SIMM.
        btr bx,bp             ;Reset the current SIMM bit in the map.
        mov al,CMSMTPL+NMIOFF
        out CMOSAD,al
        mov al,bl
        out CMOSDT,al           ;Disable the SIMM in the CMOS map.
        jmp findSIMM            ;Look for the next good SIMM.

; Try to erase the 8k boot sector to see if it is jumpered for program/erase.
erase_flash:    mov eax,' e8K'
        mov dx,SMARTVU
        out dx,eax

        mov dx,slob_extRom          ;FLASH/UVPROM enable register.
        in  al,dx
        and al,NOT slob_extRom_romEn ;Enable FLASH.
        out dx,al

        call    enable_fProgram     ;Enable flash programming.

        mov edi,flash_8kboot        ;Erase 8k boot sector.
        mov eax,'fE8K'          ;failure erasing 8k boot sector.
        call    erase_sector
        jnc eSVbios             ;If not, erase S&V bios.
        mov ax,GDTD_1M_RAM
        mov ss,ax
        jmp JROM2post           ;If none, do POST.

; Erase SBIOS & VBIOS.
eSVbios:    mov eax,'eBIO'
        mov dx,SMARTVU
        out dx,eax
        mov edi,flash_SBIOS         ;Erase SBIOS sector.
        mov eax,'fESB'          ;failure erasing SBIOS.
        call    erase_sector
        jc  do_reboot           ;Display error message.

; Program the flash SBIOS and VBIOS.
        mov eax,'pBIO'
        mov dx,SMARTVU
        out dx,eax
        mov edi,flash_SBIOS         ;Destination (start address of FLASH).
        mov esi,100000h         ;Source data in RAM @ 1 meg.
        mov ecx,18000h          ;Program 64k SBIOS and 32k VBIOS.
        mov eax,'fPSV'          ;failure Programming S & V BIOSes.
        call    program_sector
        jc  do_reboot       ;Display error message.

; Program the flash 8k boot.
        mov eax,' p8K'
        mov dx,SMARTVU
        out dx,eax
        mov edi,flash_8kboot        ;Destination (start address of 8k boot sector).
        mov esi,11e000h         ;Source data in RAM @ 1meg + 120k.
        mov ecx,2000h           ;Program 8k boot sector.
        mov eax,'fP8K'      ;failure Programming 8K boot.
        call    program_sector
        jc  do_reboot       ;Display error message.

; Flash programming successful. Cycle 'FLSH' and 'DONE' to the SmartVu
; on a 2 second cycle while waiting for a power-cycle.
        mov ax,GDTD_1M_RAM
        mov ss,ax       ;Point stack at code seg.
flash_prog_ok:  mov eax,'FLSH'
        mov cx,2        ;2 second delay.
        mov di,OFFSET fdone_msg
        jmp flash_msg   ;Display 'FLSH' and wait for 2 seconds.
fdone_msg:  mov eax,'DONE'  ;DONE message.
        mov cx,2        ;2 second delay.
        mov di,OFFSET flash_prog_ok
        jmp flash_msg

JROM2post:  mov dx,slob_extRom          ;FLASH/UVPROM enable register.
        in  al,dx
        or  al,slob_extRom_romEn        ;Enable UVPROM
        out dx,al
        jmp back2RM

bootUVPROM  ENDP
```

Procedure *BOOTFLASH*

Procedure *bootFlash* contains the 8K power-on logic executed when the system boots from the Flash.

```
The steps are as follow:
          - Checksum the system and video BIOS (96K).
          - If checksum is good, go to I.
          - If checksum is bad, go to II.
      I. - Does the user request a BIOS upgrade?
          - If upgrade is selected, go to II.
          - If upgrade is not selected, we are done with this
            routine and shall go to do regular POST.
     II. - Reset the upgrade option to prevent an endless loop.
          - Initialize a lot of stuff.
                    - Initialize 106 so that we have SmartVu.
                    - Slob initialization.
                    - Trane initialization and get at least 2 meg
                      of DRAM.
                    - Initialize the refresh counter.
                    - Set refresh page to 0 through the DMA
                      controller.
                    - Copy code to RAM and execute from RAM.
          - Set up stack.
          - Test and initialize timers.
          - Calibrate count before first call to DLY100.
          - Set up the DMA controllers.
          - Set up the interrupt descriptor table.
          - Set up the interrupt controllers.
          - Initialize the floppy subsystem.
          - Copy the BIOS from floppy.
```

- If unsuccessful, display error message and halt.
- Make sure we indeed have a valid BIOS in RAM.
- Set the Flash up for programming.
- Copy the BIOS from RAM to Flash.
- If unsuccessful, display error message and halt.

Inputs:

Native-mode, protected-mode, Trane has grabbed its I/O address.

Outputs:

Either halts or go to do regular POST.
If the BIOS checksum is good and no upgrade is requested, this routine will jump to do regular POST.
If the BIOS checksum is bad or an upgrad is requested, then this routine will halt at the end.

```
;-----------------------------------------------------------

    public  bootFlash

bootFlash proc  near

;Initialize the 106 so we can have SmartVu.
    mov sp,offset vl106_retptr
    jmp $init_vl106
vl106_retptr    dw  offset vl106_ret
vl106_ret:
    mov eax,'Dell'      ;Announce ourselves.
    mov dx,SMARTVU
    out dx,eax

;Checksum the system and video BIOS.
    mov ax,GOTO_4G      ;Flat addressing.
```

```
        mov ds,ax
        cmp dword ptr ds:[0FFFFEE076h],'IleO'
        jne badChksum

    ;Do not do the byte checksum since we have the dword XOR.
    if 0
        xor ecx,ecx
        mov cl,byte ptr ds:[0FFFF0002h];Video ROM size in 512-byte blocks.
        cmp cl,40h          ;Greater than 32k.
        ja  badChksum       ;Jump if greater than 32k.
        or  cl,cl           ;Check for 0.
        jz  short badChksum      ;Jump if 0.
        shl ecx,9           ;Convert to bytes.
        add ecx,64*1024     ;Add the size of the system ROM.
        mov ah,0            ;Initialize checksum.
        mov esi,0FFFE0000h       ;Initialize pointer to (4G-128K).
        cld             ;Set to increment.
    chkSumLoop:
        ADSIZE
        lodsb               ;al has value.
        add ah,al           ;Add value to ah.
        ADSIZE
        loop    chkSumLoop

        or  ah,ah           ;BIOS checksum good?
        jnz short badChkSum      ;Jump if bad checkum.
    endif   ;0

    ;Check the XOR of the 96K BIOS.
        mov ebx,dword ptr ds:[0FFFFEE842h];Get the system BIOS version.
        mov cx,50h/4        ;Do 20 dwords.
        mov esi,0FFFE0000h
    xorLoop1:
        ADSIZE
        lodsd
        xor ebx,eax
        loop    xorLoop1

        mov cx,(96*1024/4)-22    ;Do 24K dwords - 22 dwords.
        mov esi,0FFFE0058h
    xorLoop2:
      ADSIZE
      lodsd
      xor ebx,eax
      loop    xorLoop2

      cmp ebx,dword ptr ds:[0FFFE0050h]
      jne short badChkSum       ;Jump if bad XOR.

goodChkSum:
      mov eax,'CMOS'
      mov dx,SMARTVU
      out dx,eax

;Now that the BIOS is not corrupted, check if the user requests a BIOS upgrade.
;First, make sure we have not lost battery power to the CMOS RAM.
      mov al,CMDST+NMIOFF       ;Read CMOS status register D.
      out CMOSAD,al
      waforio
      in  al,CMOSDT
      test    al,80h            ;Did we loose battery power?
      jz  back2RM           ;Jump if we lost power.

;Second, make sure we have correct CMOS checksum.
      mov bx,2E10h+8080h        ;bh = first location not to checksum.
                        ;bl = first location to checksum.
      xor ax,ax             ;Contains the running sum.
      mov cx,ax             ;Temporary storage.
chkCMOSLoop:
      mov al,bl             ;Get location.
      out CMOSAD,al         ;Read from CMOS.
      waforio
      in  al,CMOSDT
      add cl,al             ;Tabulate word checksum.
      adc ch,0
      inc bl            ;Next location.
      cmp bl,bh             ;Done with checksum?
      jne short chkCMOSLoop     ;Jump if not done yet.

      mov al,2Eh+NMIOFF         ;Read from high byte checksum.
      out CMOSAD,al
      waforio
```

```
    in  al,CMOSDT
    cmp al,ch           ;Correct high byte checksum?
    jne back2RM         ;Jump if CMOS is corrupted.

    mov al,2Fh+NMIOFF         ;Read from low byte checksum.
    out CMOSAD,al
    waforio
    in  al,CMOSDT
    cmp al,cl           ;Correct low byte checksum?
    jne back2RM         ;Jump if CMOS is corrupted.

;Now, we are ready to inspect the "upgrade" indicators.
    mov al,CMUPGD+NMIOFF      ;Read "upgrade" indicators.
    out CMOSAD,al
    waforio
    in  al,CMOSDT
    cmp al,55h          ;Upgrade requested?
    jne back2RM         ;Jump if upgrade not requested.

Upgrade:
badChkSum:
;If you reach this point, that means you either have a corrupted BIOS or the
;user requests to upgrade the BIOS.  In either case, the following code
;applies to both.

;Reset the "upgrade" indicators to prevent an endless loop.
    mov al,CMUPGD+NMIOFF      ;Reset "upgrade" indicators.
    out CMOSAD,al
    waforio
    mov al,0
    out CMOSDT,al

;Initialize Trane and grab at least 2M of DRAM, and also initialize Slob.
    mov sp,OFFSET initStuff_retptr;Point sp to near return address.
    jmp initStuff       ;Go and do the initialization.
initStuff_retptr dw OFFSET initStuff_ret  ;Return address.

initStuff_ret:

;Set up a 32K stack at 080000h.
    mov eax,'zTAK'
    mov     dx,SMARTVU
    out dx,eax

    mov ax,GDTD_STACK
    mov ss,ax
    mov sp,8000h

;Set up the interrupt descriptor table.
    OPSIZE
    lidt    fword ptr cs:idt_ptr
;   db  2Eh,0Fh,01h,1Eh
;   dw  (OFFSET idt_ptr)+0E000h

;Go to real mode to do the floppy stuff.
getreal:
    mov eax,'rMod'
    mov dx,SMARTVU
    out dx,eax
    call    rMode

;Initialize timer and calibrate speed.
    call    init_timer
    call    calb_spd

;Set up the DMA controllers.
    mov eax,'iDMA'
    mov     dx,SMARTVU
    out dx,eax

    call    setupDMA
    jnc short setupDMA_ret

    mov eax,'xDMA'      ;DMA controller failure.
    jmp do_reboot

setupDMA_ret:
;Set up the Interrupt controllers.
    mov eax,'iPIC'
    mov     dx,SMARTVU
    out dx,eax
    call    setupPIC

setupPIC_ret:
;Initialize the floppy subsystem.
    mov dx,slob_portYY       ;Read portYY.
    in  al,dx             ;Read it.
    waforio
    or  al,slob_portYY_floppyEn  ;Turn on on-board floppy.
    out dx,al
    waforio

    mov eax,'iFDC'
    mov      dx,SMARTVU
    out dx,eax

    call    initFd           ;Do the floppy initialization.
    jnc short doCopy

    mov eax,'xFDC'      ;Floppy drive controller failure.
    jmp do_reboot

;Copy the new BIOS from floppy.
doCopy:
    call    copy             ;Copy the BIOS from the floppy.
    jnc short getProtected   ;Jump if no error in copying.

    mov eax,'xfCP'      ;Floppy copy error.
    jmp do_reboot

;Go to real mode to do the floppy stuff.
getProtected:
    mov eax,'pMod'
    mov dx,SMARTVU
    out dx,eax
    call    pMode2

;Check if we indeed have a semi-legitimate BIOS.
    mov eax,'SANE'
    mov dx,SMARTVU
    out dx,eax
    call    sanityCheck
    jnc programIt

    mov eax,'xSAN'       ;We co not have a legi BIOS.
    jmp do_reboot

;Program the Flash with the new BIOS.
programIt:
    call    program          ;Go and program the Flash.
    jc  short bootFlashError

    mov cx,800h          ;Set frequency.
    mov di,offset beep_ret2
    jmp beep
beep_ret2:

    mov ax,GDTD_64K_RAM      ;Set ss = cs.
    mov ss,ax
    jmp flash_prog_ok        ;Display 'END' message.

bootFlashError:
    mov eax,'xPRG'
    jmp do_reboot

bootFlash endp
```

Other Procedures

The actual assembly language realization used contains numerous other procedures, which will now be detailed. Most of these procedures are completely conventional. However, any of these procedures which have any particular relevance to the claimed inventions are actually listed below.

INITSTUFF

This procedure initializes the "SLOB" and "TRANE" chips.

```
;-------------------------------------------------------
   public  initStuff
initStuff proc  near
; ----- Initialize SLOB. -----
   mov eax,'iSLB'
   mov dx,SMARTVU
   out dx,eax
;
; Initialize slob_configA (0CA2h).
;  bit0:3 --> CPUtype.
;  bit4   --> 8742 installed.
;  bit5   --> NVRAM installed.
;  bit6   --> Password.
;  bit7   --> Lower bay installed. (Ignore for proto BIOS and desktop)
;
   mov dx,slob_configA      ;Read configA.
   in  al,dx
   and al,slob_configA_nvRamIn+slob_configA_8742in;Isolate the bits.
   out dx,al               ;bit0:3 and bit7 is read-only.
                        ;bit6 is written with a 0,
                        ;  this should not affect the state
                        ;  of the password.
                        ;bit4 - 8742 installed.
                        ;  if read is 0, 8742 is installed,
                        ;    then write 0 to tell the 106
                        ;    to disable its own kyb cntrl.
                        ;  if read is 1, 8742 not installed,
                        ;    then write 1 to tell the 106
                        ;    to enabled its own kyb cntrl.
                        ;bit5 - NVRAM installed.
                        ;  if read is 0, NVRAM is installed,
                        ;    then write 0 to tell SLOB to
                        ;    disable FLASH.
                        ;  if read is 1, NVRAM not installed,
                        ;    then write 1 to tell SLOB to
                        ;    enable FLASH.
;
; Initialize slob_portYY (0CA5h).
; ON RESET:
;   bit0 = 0 --> Floppy disabled.
;   bit1 = 0 --> IDE interrupt disabled.
;   bit2 = 1 --> Primary hd cntrl.
;   bit3 = 0 --> Mono.
;   bit4 = 1 --> Gate A20 set.
;   bit5 = 1 --> Kyb command enabled, no intercept.
;   bit6 = 1 --> BIOS reset on.
;   bit7 = 0 --> Relay off. (Ignore for desktop)
;
    mov dx,slob_portYY       ;Read portYY.
    in  al,dx
    and al,not (slob_portYY_gateA20+slob_portYY_kA20cmdEn+slob_portYY_biosRstDrv)
                        ;Reset A20.
                        ;Enable kyb command intercept.
                        ;BIOS reset off.
    out dx,al
;
; Initialize slob_portXX (0CA6h).
; ON RESET:
;   bit0 = 1 --> RSTNMI inactive (read-only).
;   bit1:2 = 0 --> Speaker off.
;   bit3 = 0 --> FLASH program/erase disabled.
;   bit4 = 1 --> Native mode.
;   bit5 = 0 --> SMVU reset enabled.
;   bit6 = 0 --> Mouse IRQ enabled.
;   bit7 = 0 --> Flush disabled.
;
    mov dx,slob_portXX       ;Write portXX.
    mov al,01010111b         ;bit7 = disable flush.
    out dx,al             ;bit6 = mouse IRQ disabled.
```

```
                          ;bit5 = sm/u reset enabled.
                          ;bit4 = native mode.
                          ;bit3 = flash erase/program disabled.
                          ;bit2:1 = speaker on HIGH.
                          ;bit0 read only (RSTNMI).
;
; Initialize slob_portZZ (0CA7h).
; ON RESET:
;   bit0 = 0 --> VGA disabled.
;   bit1 = 0 --> VGA reset on.
;   bit2 = 0 --> VGA IRQ9 disabled.
;   bit3 = 0 --> IDE SLVACT masked.
;   bit4 = 1 --> don't cares (read-only).

    mov dx,slob_portZZ        ;Write portZZ.
    mov al,11110001b          ;VGA enabled.
    out dx,al
    mov al,11110011b          ;VGA reset off.
    out dx,al
;
;Don't need to initialize cpu types; POST will (0CA2h and 0CA3h).
;
;
; Initialize the phantom counter control register, slob_pccr (0CA0h).
;
;wrc???

;
; Initialize slob_power_good_mask (0CA1h).
;This register does not need initialization; defaults to
; 3Fh.  This register sets the time for stabling power output
; for a tower system when the relay to the lower drive bay kicks in.
;

;
; Initialize slob_config8 (0CA3h).
; This register does not need initialization.
;

;
; Initialize slob_extRom (0CA4h).
; This register does not need initialization.
;

; ------ Initialize RAM -----
;
    mov eax,'iRAM'
    mov dx,SMARTVU
    out dx,eax

        mov bp,OFFSET initRAM_ret;Point sp to near return address.
        jmp RAM_init         ;Initialize Trane.

initRAM_ret:
;Map in the first available SIMM at 0-2Mb.
        mov al,CMSMTPL+NMIOFF
        out CMOSAD,al
        in  al,CMOSDT        ;Get SIMM map from CMOS.
        movzx   bx,al            ;Save SIMM configuration in bx.
        mov bp,es
        shl ebp,16           ;ES into hi word of eax.
        mov bp,ds            ;Save ds.
        mov ax,GDTD_4G       ;4G data seg ...
        mov ds,ax            ;into ds.
        mov es,ax            ;into es.
findSIMM2:  bsf dx,bx            ;Find first installed SIMM position.
        jnz SHORT mapSIMM2       ;If a SIMM is found, map it in @ 0.
        mov eax,'fRAM'       ;Can't find any RAM.
        mov cx,0             ;Delay forever.
        jmp flash_msg        ;Display error message.

mapSIMM2:   TRANE_OUT_IX     TRANE_RCROM,dl   ;Program 1st SIMM at 0 as a 256kbit part.

; Copy 8k boot from ROM to RAM at 11e000h (1M + 128k - 8k).
        mov ecx,2000h SHR 2      ;8kb == 2k DWORDs.
        mov edi,10000h       ;Destination in RAM above 1M.
        mov esi,0ffffe000h       ;Source in 4G ROM.
        cld              ;Forward MOVSD.
        ADSIZE
        rep movsd            ;move 8k boot from 4G ROM to RAM above 1M.

; Check the RAM code against ROM to verify the copy and validate RAM
```

```
        mov ecx,2000h SHR 2      ;8kb -- 2k DWORDs.
        mov edi,10000h          ;Destination string in RAM above 1M.
        mov esi,0ffffe000h       ;Source string in 4G ROM.
        ADSIZE
        repe    cmpsd            ;Search for non-matching DWORD.
        je  SHORT jmp2RAM2 ;If RAM/ROM match then jump to the RAM code.

; If the 8k boot in RAM is corrupted then assume that SIMM is bad and
; mark it out in CMOS. Go back and look for the next good SIMM.
        btr bx,dx               ;Reset the current SIMM bit in the map.
        mov al,CMSMTPL+NMIOFF
        out CMOSAD,al
        mov al,bl
        out CMOSDT,al           ;Disable the SIMM in the CMOS map.
        jmp findSIMM2           ;Look for the next good SIMM.

; Jump into the 8k boot code that has been copied to RAM.
jmp2RAM2:   mov ds,bp            ;Restore ds.
        shr ebp,16
        mov es,bp
        JMPP    GDTC_64K_RAM        ;Load cs with selector for 1M RAM code.

;Relocate the GDT.
    OPSIZE
    lgdt    fword ptr cs:ram64_gdt_ptr

initStuffDone:
    ret

initStuff endp
```

RAM_init

Procedure *RAM_init* performs RAM initialization for the 8k boot code, initializes TRANE, identifies memory, initializes it, turns on refresh and points RCR0 to a block from 0 - 2Mb.

Inputs:

HSM_xx ... Host state machine initialization tables.

FS ... Must point to a 4Gb data selector based @ 0.
bp contains the return address.
NOTE: It doesn't matter what type of SIMM we have identified. The presence test assumes that it's a 256kbit chip and initializes the RCR accordingly. If it's actually a 1Mbit or a 4Mbit SIMM it should still behave ok if it's programmed as a 256kbit. Since all we need is 128kb of RAM for copying the UVPROM, who cares what size the SIMM really is?

Outputs:

SUCCESS ... If successful, returns the following:

- All SIMMs identified as present or not and tabulated in CMSMTPL below.

  CMSMTPL ... CMOS byte containing the SIMM map.

- RCR0M points to 2 Mb of DRAM. SIMMs for this RAS programmed as 256kbit devices.

- Refresh enabled.

FAILURE ... If no RAM can be found, issues 'fRAM' to SmartVu and waits forever.

$INIT_VL106

Procedure *$init_vl106* initializes the vl106. The VL106 is a chip which, among other functions, replaces the 8742 keyboard handler.)

PMODE

This procedure is a Protected mode entry routine.

Inputs:

DS ... points to the segment containing "gdt_ptr" and NULL_IDT.

RMODE

This routine is created for the floppy code to run in real mode.

Inputs:	none.
Outputs:	real mode.

PMODE2

This routine is created for the floppy code. This will switch the execution back to protected after the floppy code is executed. **[Uses different IDT and GDT tables ub low RAM.]**

Inputs:	none.
Outputs:	protected mode.

CMOSSUM

This routine checksums the CMOS range 10h thru 2Dh.

Inputs:

DI ... contains NEAR return offset.

Outputs: none.

FLASH_MSG

Procedure *flash_msg* displays the SMARTVU message in EAX, waits

CX seconds and returns to the NEAR address in DI.

Inputs:

SS ... Writable segment pointing to code.

DI ... Contains the NEAR return address.

CX ... The number of seconds to delay after displaying the SMARTVU message.

EAX ... Contains the SmartVu message.

DELAY

Procedure *delay* waits for CX seconds. This procedure uses the RTC to implement a delay specified by the count in seconds in CX. If the RTC battery is bad, we just execute a fixed delay loop of DUMMY_DELAY iterations.

Inputs:

cx ... Countains the number of seconds to delay.

GET_SEC

This routine reads the seconds count from the RTC (real-time clock), and waits if a time update is in progress.

Inputs:

SI ... Contains the NEAR return address.

Outputs:

AL ... Contains the RTC seconds count.

BEEP

This routine beeps!

Inputs:

cx = frequency.

di = near return address.

Outputs:

none.

INIT_TIMER

This routine tests and initializes the timers.

SETUPDMA

Procedure *setupDMA* will set up the DMA controllers just like the regular POST. The code are identical so as not to introduce any problem with the INT13H routine.

Inputs:

Native mode, protected mode.

Outputs:

If error, let regular POST reports the error.

If no error, DMA controllers are now ready for the INT13H routine.

SETUPPIC

Procedure *setupPIC* will set up the PICs just like the regular POST. The code is identical to that used in the POST, so as not to introduce any problem with the INT13H routine.

Inputs:

Native mode, protected mode.

Outputs:

The PICs are now ready for the INT13H routine.

COPY

Procedure *copy* copies the new BIOS from the floppy to RAM. The steps are as follow:

- Find out if file exists. The file has to be on a 1.2M floppy in the root directory with a filename of "DELLBIOS.BIN".

Inputs:

none.

Outputs:

Carry flag set if error.

```
;----------------------------------------------------------

    public  copy

copy    proc    near

    mov eax,'ROOT'
    mov dx,SMARTVU
    out dx,eax

    call    fileExist       ;Find out if file exists?
    jnc short gotRoot       ;Jump if file exists.

    mov dx,fdcDOR       ;Access fdcDCR.
    mov al,0Ch          ;Turn off motor, disabled interrupts
                    ;  and DMA.
    out dx,al

    mov eax,'xROM'
    jmp do_reboot

gotRoot:
    push    eax
    push    dx
```

```
        mov eax,'COPY'
        mov dx,SMARTVU
        out dx,eax
        pop dx
        pop eax

        call    copyFile           ;Go and try to copy file.
                      ;  Return with carry set if error.
        mov dx,fdcDOR       ;Access fdcDOR.
        mov al,0Ch          ;Turn off motor, disabled interrupts
                      ;  and DMA.
        out dx,al
        jnc short exitt     ;Exit if no error from copyFile?

        mov eax,'xCPY'
        jmp do_reboot

exitt:
        ret

copy    endp
```

FILEEXIST

Procedure *fileExist* reads in the root directory, and searches to see whether file "DELLBIOS.BIN" is in the directory.

Inputs:

none.

Outputs:

Carry clear if file exists.

ax has the first disk cluster if file exists.

bx:dx has file size.

SETTYPE

Procedure *setType* determines the type of floppy drive and sets the drive parameters accordingly.

Inputs:

none.

Outputs:

Parameters are initialized.

COPYFILE

Procedure *copyFile* copies the desired file from drive 0 to RAM.

The steps are as follow:

- Make sure the file size is what we expected.
- Read the FAT table.
- Copy the file to DRAM.

Inputs:

ax has the first disk cluster of the file.

bx:dx has file size.

es=GDTD_BUFFER.

Outputs:

Carry flag set if error.

SANITYCHECK

Procedure *sanityCheck* will try to verify that the BIOS copied is a legitimate Dell BIOS.

Inputs:

BIOS (112K) spans from 5000:0000 thru 6000:C000.

Outputs:

Carry set if we do not have a legi BIOS.

```
;-------------------------------------------------------
    public  sanityCheck

sanityCheck proc near

    push    ds
    push    eax
    push    ebx
    push    ecx
    push    esi

;Check the Dell signature.
    cmp dword ptr ds:[05E076h],'lleD'
    jne illegitimate

;Do not byte checksum the BIOS since we have dword XOR.
if 0
;Checksum the system and video BIOS.
    xor ecx,ecx
    mov cl,byte ptr ds:[060002h];Video ROM size in 512-byte blocks.
    shl ecx,9           ;Convert to bytes.
    add ecx,64*1024     ;Add the size of the system ROM.
    mov ah,0            ;Initialize checksum.
    mov esi,050000h     ;Initialize pointer.
    cld             ;Set to increment.
saniLoop:
    ADSIZE
    lodsb               ;al has value.
    add ah,al           ;Add value to ah.
    ADSIZE
    loop    saniLoop

    or  ah,ah           ;BIOS checksum good?
    jnz short illegitimate      ;Jump if bad checkum.
endif   ;0

;Check the XOR of the 96K BIOS.
    mov ebx,dword ptr ds:[05E842h];Get the system BIOS version.
    mov cx,50h/4        ;Do 20 dwords.
    mov esi,050000h
saniLoop1:
        ADSIZE
        lodsd
        xor ebx,eax
        loop     saniLoop1

        mov cx,(96*1024/4)-22    ;Do 24K dwords - 22 dwords.
        mov esi,050058h
    saniLoop2:
        ADSIZE
        lodsd
        xor ebx,eax
        loop     saniLoop2

        cmp ebx,dword ptr ds:[050050h]
        jne short illegitimate   ;Jump if bad XOR.

    saniExit:
        pop esi
        pop ecx
        pop ebx
        pop eax
        pop ds

        ret

    illegitimate:
        stc
        jmp short saniExit

    sanityCheck endp
```

NEXT

Procedure *next* gets the next link from a 12-bit FAT.

Input:

ax = current entry number.

Output:

ax = next element in the chain.

REL2ABS

Procedure *rel2abs* converts the relative sector number to the absolute sector location.

Input:

ax = relative sector number.

Output:

ch = track number.

cl = sector number.

dh = head number.

dl = 0 = drive number.

GETFAT

Procedure *getFat* reads in the FAT table.

Inputs:

es = GDTD_BUFFER.

Outputs:

Carry set if error.

If successful, FAT resides from 4000:0000 thru 4000:0E00.

PROGRAM

Procedure *program* writes data into the flash ROM.

Inputs:

none.

Outputs:

Carry set if error.

;---

```
    public   program

program proc     near

    push     ds
    push     es

    mov ax,GDTD_4G        ;Flat address.
    mov ds,ax
    mov es,ax

    call     enable_fProgram       ;Enable flash programming.
;   call     setNCA           ;Set up one NCA descriptor.

;Do the 112k BIOS and video sector.
    mov esi,50000h        ;Offset of buffer.
    mov edi,0FFFE0000h       ;Start of sector
    mov ecx,1C000h        ;112k.

    push     eax
    mov eax,'ERAZ'
    mov dx,SMARTVU
    out dx,eax
    pop eax

    call     erase_sector
    jc  short bad_flash

    push     eax
    mov eax,'PROG'
    mov dx,SMARTVU
    out dx,eax
    pop eax

    call     program_sector
    jc  short bad_flash

    call     disable_fProgram         ;Disable flash programming.
    clc

flash_done:
    pop es
    pop ds

    ret

bad_flash:
    call     disable_fProgram         ;Disable flash programming.
    stc
    jmp short flash_done

program endp
```

ERASE_SECTOR

Procedure *erase_sector* performs the following steps:

- Clear the status register.
- Set the FLASH to read status mode.
- Set the FLASH for erase.
- Do the erase.
- Verify the erase is successful by reading the status register.
- Set the carry flag if erase unsuccessful.
- Clear the status register.
- Set the FLASH to read array mode.

Inputs:

es:edi = address of sector to be erased.

FLASH program/erase better be enabled before calling this routine.

Outputs:

If successful, carry flag cleared.

;

```
; WARNING!!! -- The FLASH program jumper must be set for program/erase.
;
;------------------------------------------------------

    public  erase_sector

erase_sector proc near

    push    esi         ;Save esi.
    push    edi         ;Save edi.
    push    ecx         ;Save cx.
    push    ax          ;Save ax.
    push    dx          ;Save dx.

    call    clr_Flash_status

    push    ecx
    mov cx,300
    call    usecWait
    pop ecx

    mov byte ptr es:[edi],flash_erase_setup;Get ready to erase.

    push    ecx
    mov cx,300
    call    usecWait
    pop ecx

    mov byte ptr es:[edi],flash_erase_go;Do the erase.

    push    ecx
    mov cx,300
    call    usecWait
    pop ecx

    mov ecx,0           ;Software timeout.
erase_sector_wait:
    call    read_Flash_status   ;Put status in al.
    test    al,flash_status_wsm_busy;Erase finish yet?
;   jz  erase_sector_wait
```

```
        loopz   erase_sector_wait    ;Jump if not done.

        jz  short erase_sector_err   ;Jump if times out.

        test    al,flash_status_erase_fail+flash_status_vpp_low;Erase fail?
        jnz short erase_sector_err   ;Jump if erase fail.

    clear_erase_sector_status:
        call    clr_Flash_status
        mov byte ptr es:[edi],flash_read;Set the FLASH to read array mode.

        clc

    erase_sector_done:
        pop dx          ;Restore dx.
        pop ax          ;Restore ax.
        pop ecx              ;Restore cx.
        pop edi         ;Restore edi.
        pop esi         ;Restore esi.

        ret

    erase_sector_err:
        call    clr_Flash_status
        mov byte ptr es:[edi],flash_read;Set the FLASH to read array mode.

        stc
        jmp short erase_sector_done

    erase_sector endp
```

PROGRAM_SECTOR

Procedure *program_sector* writes a sector of the flash ROM.

Inputs:

ecx = number of bytes to program.

ds:esi points to beginning of data.

es:edi points to beginning of sector.

The steps are as follow:

- Clear the status register.
- Set the FLASH to read status mode.
- Go and program the sector.
- Abort if error with the carry flag set.
- Clear the status register.
- Set the FLASH to read array mode.

Outputs:

If successful, carry flag cleared.

```
;
; WARNING!!! -- The FLASH program jumper must be set for program/erase.
;
;-------------------------------------------------

    public  program_sector

program_sector proc near

    push    eax         ;Save ax.
    push    ecx         ;Save cx.
    push    esi         ;Save esi.
    push    edi         ;Save edi.

    call    clr_Flash_status

    cld

from_esi:
;ecx - byte counts.
    mov byte ptr es:[edi],flash_program_setup;Get ready to program.
    mov al,byte ptr ds:[esi]
    mov byte ptr es:[edi],al

    push    ecx
    mov cx,5
    call    usecWait
    pop ecx

    inc esi
    inc edi

    ADSIZE
    loop    from_esi

    mov ecx,0            ;Software timeout.
program_new_wait:
    call    read_flash_status       ;Put status in al.
    test    al,flash_status_wsm_busy;Program finish yet?
;   jz  short program_new_wait
    loopz   program_new_wait     ;Jump if not done.

    jz  short program_sector_err ;Jump if times out.

    test    al,flash_status_prog_fail+flash_status_vpp_low;Program fail?
    jnz short program_sector_err ;Jump if program fail.

program_sector_ok:
    call    clr_Flash_status
    mov byte ptr es:[edi],flash_read;Set the FLASH to read array mode.

    clc

program_sector_done:
    pop edi          ;Restore edi.
    pop esi          ;Restore esi.
    pop ecx               ;Restore cx.
    pop eax          ;Restore ax.

    ret

program_sector_err:
    call    clr_Flash_status
    mov byte ptr es:[edi],flash_read;Set the FLASH to read array mode.

    stc
    jmp short program_sector_done

program_sector endp
```

ENABLE fPROGRAM

Procedure *enable fProgram* enables program/erase operations on the FLASH ROM.

Inputs: none.

Outputs:

FLASH is ready to be programmed or erased.

```
;------------------------------------------------

enable_fProgram proc near

    push    ax          ;Save ax.
    push    dx          ;Save dx.

    mov dx,slob_portXX       ;Access portXX.
    in  al,dx           ;Read in current value.
    waforio
    or  al,slob_portXX_flhPrg    ;Enable FLASH program/erase.
    out dx,al           ;Write out the new value.

    pop dx          ;Restore dx.
    pop ax          ;Restore ax.

    ret

enable_fProgram endp
```

DISABLE fPROGRAM

Procedure *disable fProgram* disables FLASH program/erase.

Inputs: none.

Outputs: FLASH cannot be programmed or erased.

```
;---------------------------------------------

disable_fProgram proc near

    push    ax          ;Save ax.
    push    dx          ;Save dx.

    mov dx,slob_portXX       ;Access portXX.
    in  al,dx           ;Read in current value.
    waforio
    and al,not slob_portXX_flhPrg;Disable FLASH program/erase.
    out dx,al           ;Write out the new value.

    pop dx          ;Restore dx.
    pop ax          ;Restore ax.

    ret

disable_fProgram endp
```

USECWAIT

Procedure *usecWait* creates cx microseconds of delay.

Inputs:

cx = number of usec.

Outputs:

cx destroyed.

CLR_FLASH_STATUS

Procedure *clr_Flash_status* clears the FLASH status register. This routine was created because the new spec requires the address to be at 00000h of the Flash.

Inputs:	none.
Outputs:	Flash status register cleared.

```
;-----------------------------------------------
clr_Flash_status proc near
;NOTE:  Do not need to save the entry mode.  It will always be native mode.

    push    ax
    push    ds
    push    ecx

    mov ax,GDTD_4G       ;4Gb descriptor.
    mov ds,ax

    mov byte ptr ds:[0FFFFE000h],flash_clear_status

    mov cx,15
    call    usecwait

    pop ecx
    pop ds
    pop ax

    ret

clr_Flash_status endp
```

READ_FLASH_STATUS

Procedure *read_Flash_status* reads the FLASH status register at 00000h of the Flash.

Inputs:	none.
Outputs:	Flash status register read in al.

```
;---------------------------------------------------

read_Flash_status proc near

;NOTE:  Do not need to save the entry mode.  It will always be native mode.

    push    ecx
    push    ds
```

```
        mov ax,GDTD_4G        ;4Gb descriptor.
        mov ds,ax

;       FLUSH_CACHE

        mov byte ptr ds:[0FFFE000Ch],flash_status
        mov cx,15
        call    usecWait
        mov al,byte ptr ds:[0FFFEC000h]
        mov cx,15
        call    usecWait

        pop ds
        pop ecx

        ret

read_Flash_status endp
```

<u>*NMI_ISR*</u>

Procedure *NMI_ISR* is a dummy interrupt handler, which swallows any NMMI interrupt.

<u>*DMATST*</u>

This routine tests the DMA address and word count registers.

Input:

CX = # of ports to test

DX = first port to test

SI = increment between ports

Output: flags ?

AL = ?

AH = ?

CX = 0

CALB_SPD

Procedure *CALB_SPD* ("calibrate speed") sets DLLY_CNT with count for 100 us

FDISR

Procedure *fdISR* is the floppy drive controller interrupt handler. The steps are as follow:

- Send EOI to master PIC.
- Set interrupt bit in DRVSTAT to indicate an interrupt has occurred.

INT13H

Procedure *int13h* determines and executes the desired int13h procedure functions.

RESETFLOP

Procedure *resetFlop* resets the floppy system. This is part of the INT13H procedure.

Inputs:

ah = 0

dl = drive number (0-3), bit7 = 0 for floppy.

Outputs:

Carry flag set if error.

ah = status/error code.

ds:ERRSTAT(FDATA) = ah.

READSECTOR

Procedure *readSector* reads al sectors from the floppy.

Inputs:

ah = 02h.

al = number of sectors.

ch = track number.

cl = sector number.

dh = head number.

dl = drive number.

es:bx = address of buffer.

Outputs:

Carry flag set if error.

al = number of sectors transferred, ah = status = ERRSTAT.

SETCARRY

Procedure *setCarry* sets the carry flag on the stack. This is part of the INT13H procedure.

CLRCARRY

Procedure *clrCarry* clears the carry flag on the stack. This is part of the INT13H procedure.

Inputs: none.

Outputs:

Carry flag in the flag register on the stack is cleared.

Interrupt flag also set.

DSKRESET

Procedure *dskReset* resets the floppy system. This is part of the INT13H procedure.

The steps are as follow:

- Disable interrupts.
- Clear DRVSTAT and ERRSTAT.
- Get floppy motor status.
- Issue reset command.
- Turn off reset command.
- Enable interrupts and wait for reset result interrupt.
- Check result.

Inputs:

dl = drive number (0-3), bit7 = 0 for floppy.

Outputs:

Carry flag set if error.

ds:ERRSTAT(FDATA) = ah = error code if error, else 0.

WAITINT

Procedure *waitInt* waits for a FDC interrupt. This is part of the INT13H procedure. This routine uses a regular wait instead of int15h.

Inputs: none.

Outputs:

Carry flag set if timeout error, else not.

ERRSTAT bit7 = 1 if timeout.

FDCRDY

Procedure *fdcRdy* waits for the floppy controller to be ready. This is part of the INT13H procedure.

Inputs: none.

Outputs:

Carry flag set if timeout.

SISSTAT

Procedure *sisStat* senses the interrupt status. This is part of the INT13H procedure.

Inputs: none.

Outputs:

Carry flag set if error, ERRSTAT set, ah = FDC status,
else Carry flag clear and al = present cylinder.

WRTCMD/WRTCMDA

Procedures *wrtCmd* and *wrtCmdA* write commands to the FDC, and are part of the INT13H procedure.

Inputs:

al = command.
ds points to FDATA.

Outputs:

Carry flag and ERRSTAT set if error.
ah, dx destroyed.

RSLTRD7/RSLTRD

Procedure *rsltRd7* reads 7 result bytes from the FDC. Procedure *rsltRd* reads cx result bytes from the FDC. These are part of the INT13H procedure.

Inputs:

cx = number of result bytes to read.

ds points to FDATA.

Outputs:

al = FDC status.

ah = FDC ST0, only valid if no carry.

ERRSTAT bit 7 = 1, carry set on timeout.

ERRSTAT bit 5 = 1, carry set if FDC results cannot be flushed.

Carry set if less results available than expected.

DSKST = FDC ST0, only valid if no carry.

dx non-zero.

I82077EXIST

Procedure *i82077exist* determines if a i82077 FDC is installed.

This is part of the INT13H procedure.

Inputs: none.

Outputs:

Carry flag set if i82077 installed and the FIFO enabled.

stk_tmp bit0=1 if i82077 installed and bit1=1 if FIFO enabled.

MLTCMD2/MLTCMD

Procedure *mltCmd2* writes 2 commands to the FDC. Procedure *mltCmd* writes cx commands to the FDC. These are part of the INT13H procedure.

Inputs:

cx = number of bytes to write (mltCmd).

bx = starting table offset.

Outputs:

es:si = disk parameter block.

ERRSTAT bit7=1 on timeout, bit5=1 on controller error.

Carry flag set on timeout or controller error, else not set.

DPBADR

Procedure *dpbAdr* puts the address of the disk parameter block into es:di. This is part of the INT13H procedure.

Inputs: none.

Outputs: es:si points to disk parameter block.

CONFIGURE

Procedure *configure* issues the configure command to enable the FIFO in the i82077 FDC. This is part of the INT13H procedure.

Inputs:

ah = byte 2 of configure command.

Outputs:

Carry flag set if FDC error, else cleared.

DSKMOT

Procedure *dskMot* turns the floppy drive motor on. This is part of the INT13H procedure.

Inputs:

dl = drive number.

Outputs: none.

CHKCHG

Procedure *chkChg* checks the change line on the floppy drive. This is part of the INT13H procedure.

Inputs: none.

Outputs: Carry flag set if error.

GETDRVINFO

Procedure *getDrvInfo* gets drive information from HDCFLG. This is part of the INT13H procedure.

Inputs: none.

Outputs:

Low 3 bits of al = drive information.

CLRCHG

Procedure *clrChg* tests and clear the change line. This is part of the INT13H procedure.

Inputs:

dl = drive number.

Outputs:

Carry flag set if timeout.

ERRSTAT = timeout or change line error.

DOSEEK/DOSEEK2

Procedures *doSeek* and *doSeek2* will seek to track specified by *ch*. This is part of the INT13H procedure.

Inputs:

ch = track to seek.

Outputs:

Carry flag and ERRSTAT set if error.

DRIHEDSEL

Procedure *driHedSel* loads the head (HDS) and the drive (DS1, DS0) parameters to the FDC. This is part of the INT13H procedure.

Inputs: none.

Outputs: none.

INITDMA

Procedure *initDMA* will initialize the DMA controller for floppy operations. This is part of the INT13H procedure.

Inputs: none.

Outputs:

Carry flag and ERRSTAT set if error.

HEADSETTLE

Procedure *headSettle* conditionally waits for the head to settle. This is part of the INT13H procedure.

Inputs:

dx non-zero if head settle wait needed.

Outputs:

es:si points to DPB.

SPINUP

Procedure *spinup* conditionally waits for the floppy to spinup. This is part of the INT13H procedure.

CHKTYP

Procedure *chkTyp* checks the media type. This routine is called prior to read/write operation to verify and update, if necessary, FDMED. Returns with the carry flag set if no remaining valid FDMEDs to try. This is part of the INT13H procedure.

Inputs:

FDMED, ERRSTAT (if non-zero, it is a retry).
bx = drive.

Outputs:

FDMED updated:
ERRSTAT = carry flag = 0 if any more valid rate/step combos.
FDMED restored:
FDOPER = 0.
ERRSTAT and carry flag set if no more valid rate/step combos.;

FDTYPIF

Procedure *fdTypIf* gets floppy disk type from CMOS if CMOS is valid. This is part of the INT13H procedure.

Inputs: none.

Outputs:

Carry flag set if CMOS invalid, else
al = drive type, zero flag set if no drive or type unknown,

top of ah non-zero if another drive.

FDTYPE

Procedure *fdType* gets floppy disk type from CMOS. This is part of the FDTYPIF routine, and part of the INT13H procedure.

Inputs: none.

Outputs:

al = drive type, zero flag set if no drive or type unknown,
top of ah non-zero if another drive.

SETRATE

Procedure *setRate* sets the transfer rate. This is part of the INT13H procedure.

Inputs:

bx = drive number.

Outputs:

Rate control port updated.

CHKRES

Procedure *chkRes* reads and checks the results from the FDC. This is part of the INT13H procedure.

Inputs: none.

Outputs:

Carry flag set if error.
al = number of sectors transferred.

AFTERTRY

Procedure *afterTry* sets the zero flag if there is still more retry after an unsuccessful operation. This is part of the INT13H procedure.

Inputs: none.

Outputs:

Zero flag set if there is still more retry.

RESTYP

Procedure *resTyp* forces FDMED[bx] media known and restores compatible low bits. This is part of the INT13H procedure.

Inputs:

ah = top three bits of FDMED[bx].

Outputs:

FDMED[bx] bit4 = 1.

STODRVINFO

Procedure *stoDrvInfo* puts the drive information in al into HDCFLG. This is part of the INT13H procedure.

Inputs:

al = drive information.

bx = drive number.

Outputs:

Drive information in [bx] half of HDCFLG.

INITFD

Procedure *initFD* will initialize the floppy subsystem and all the variables used by the INT13H routine. This is part of the INT13H routine.

Inputs: none.

Outputs:

DRVSTAT, FMOTS, FDTIMO, ERRSTAT, DSKST, HDCFLG, FDMED, FDOPER, WTACTF, FDRATE, are all initialized.

DSKTEST

Procedure *dskTest* will test an. determine drive type.

Inputs:

ss:bp points to stack frame for disk driver.

stk_dl[bp] = drive number.

Outputs:

FDMED[drive #], HDCFLG initialized.

Carry flag set if no drive.

SDRVST

Procedure *sDrvSt* senses the drive status.

Inputs: none.

Outputs:

Carry flag set, ERRSTAT set if error.

Carry flag clear, ah=FDC's ST0 if no error.

Zero flag set if track 0 reached.

CKFDCFG

Procedure *ckFdCfg* checks the floppy configuration data against the actual installed hardware.

Inputs:

byte ptr stk_dl[bp] = drive number.

si points to the media type for that drive number.

Outputs:

Carry set if different configuration.

CMREAD

Procedure *CMREAD* is a Subroutine to Read a CMOS register.

Input:

AL = CMOS register address to read

Output:

AL = CMOS register value

Interrupt Flag cleared

Carry Flag cleared

Zero Flag cleared.

CMWRT

This is a Subroutine to Write a CMOS register.

Input:

AH = CMOS register address to write, Bit 7 = 0 enables NMI

AL = Value to write to CMOS register

Output:

Interrupt Flag cleared

Carry Flag cleared

Zero Flag cleared.

DLY100

This procedure will delay for 100 us. The loop count for the delay function is taken from DLY_CNT. DLY_CNT is set during ATPOST, and whenever turbo speed is changed by the routine CALB_SPD. It takes no inputs and gives no outputs.

Further Modifications and Variations

It will be recognized by those skilled in the art that the innovative concepts disclosed in the present application can be applied in a wide variety of contexts. Moreover, the preferred implementation can be modified in a tremendous variety of ways. Accordingly, it should be understood that the modifications and variations suggested below and above are merely illustrative. These examples may help to show some of the scope of the inventive concepts, but these examples do not nearly exhaust the full scope of variations in the disclosed novel concepts.

In particular, the disclosed innovations are not by any means limited to DOS systems, nor to systems using 80x86 microprocessors, nor to systems using a single microprocessor as the CPU. The disclosed innovations provide generally applicable architectural techniques, which can be applied to a wide variety of computer systems, including high-performance systems and multiprocessing systems.

It should also be noted that the source for the user-supplied BIOS code does not have to be a floppy drive. For example, a tape cartridge or a dial-in telephone interface could be used instead.

In addition, although the Flash EPROM is the preferred device technology for the second nonvolatile memory, it should be recognized that other nonvolatile storage technologies can be used if they become practical. Thus, many of the innovative features of the presently preferred embodiment can be directly adapted to a system using EEPROMs, battery-backed SRAM chips, ferroelectric RAMs, or future technologies.

It should also be noted that BOTH of the boot memories can be rewriteable nonvolatile memories if desired. Even though this arrangement provides slightly less robustness than the preferred embodiment, it still provides a large added margin of safety over the use of a single rewriteable nonvolatile boot memory.

As will be recognized by those skilled in the art, the innovative concepts described in the present application can be modified and varied over a tremendous range of applications, and accordingly the scope of patented subject matter is not limited by any of the specific exemplary teachings given.

Claims

1. A computer system including a rewritable non-volatile memory which holds bootstrapping instructions including instructions capable of effecting the over-writing of instructions held in the rewritable non-volatile memory, from an external source, following a hard reset of the system.

2. A computer system including a rewritable non-volatile memory, a second non-volatile memory which holds bootstrapping instructions including instructions capable of effecting the over-writing of instructions held in the rewritable non-volatile memory from an external source, and means for selecting either the second non-volatile memory or the rewritable non-volatile memory as its source of bootstrapping instructions, following a hard reset of the system.

3. A computer system as claimed in claim 2, in which the bootstrapping instructions held in the second non-volatile memory are such as to effect the over-writing of instructions held in the rewritable non-volatile memory by the bootstrapping instructions held in the second nonvolatile memory, following a hard reset of the system.

4. A computer system as claimed in claim 2 or claim 3, which includes a user settable switch for selecting the second non-volatile memory initially as its source of bootstrapping instructions and which is capable of reversing the switch setting to select the rewritable non-volatile memory as its source of bootstrapping instructions thereafter.

5. A computer system as claimed in any one of claims 2 to 4, wherein the second non-volatile memory is a rewritable memory and includes means for preventing its data from being overwritten.

6. A computer system as claimed in any one of claims 1 to 5, including manually operable means which has a first setting for preventing the over-writing of instructions held in the rewritable non-volatile memory and a second setting for permitting the over-writing of instructions held in the rewritable non-volatile memory.

7. A computer system as claimed in any one of claims 1 to 6, including a multi-element diagnostic display capable of indicating the result of an operation to over-write instructions held in the rewritable non-volatile memory.

8. A method of operating a computer system including the steps of first writing bootstrapping instructions from a second non-volatile memory to a rewritable non-volatile memory following a hard reset of the system and, thereafter, reading the bootstrapping instructions from the rewritable non-volatile memory in bootstrapping the computer system.

9. A method of operating a computer system as claimed in claim 8, including the steps of reading initial bootstrapping instructions from the rewritable non-volatile memory in bootstrapping the computer system and overwriting the remaining bootstrapping instructions in the rewritable non-volatile memory with data from an external source.

10. A method of operating a computer system as claimed in claim 8 or claim 9, including the steps of reading bootstrapping instructions from the rewritable nonvolatile memory, subjecting the bootstrapping instructions to validity checks as they are read and, when a validity check is failed, over-writing the bootstrapping instructions held in the rewritable non-volatile memory with the bootstrapping instructions held in the second non-volatile memory.

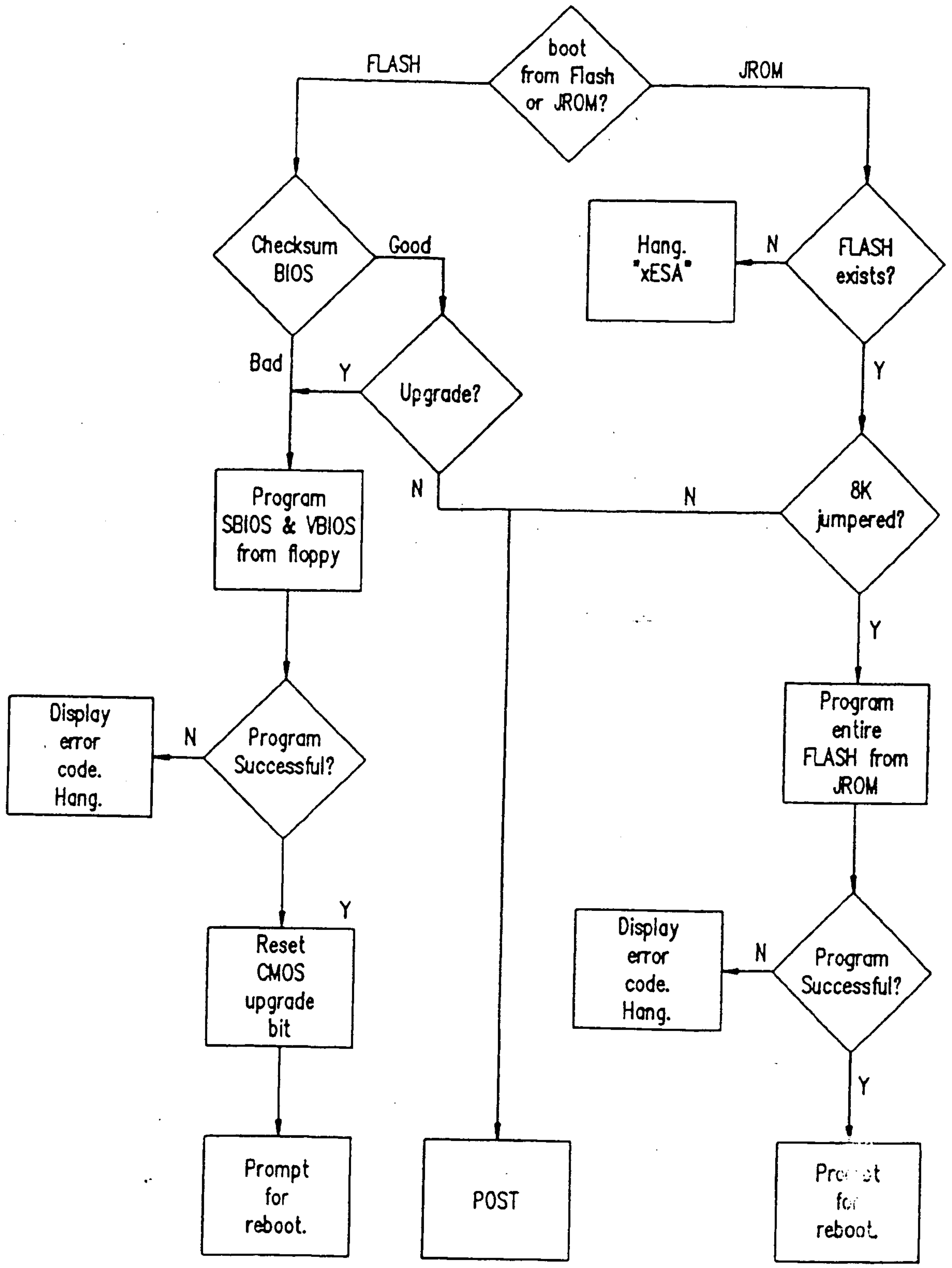
boot from Flash or JROM?
FLASH
JROM
Checksum BIOS
Good
Bad
Upgrade?
Y
N
Hang. "xESA"
FLASH exists?
N
Y
8K jumpered?
N
Y
Program SBIOS & VBIOS from floppy
Program Successful?
N
Y
Display error code. Hang.
Reset CMOS upgrade bit
Prompt for reboot.
POST
Program entire FLASH from JROM
Program Successful?
N
Y
Display error code. Hang.
Prompt for reboot.

Figure 1

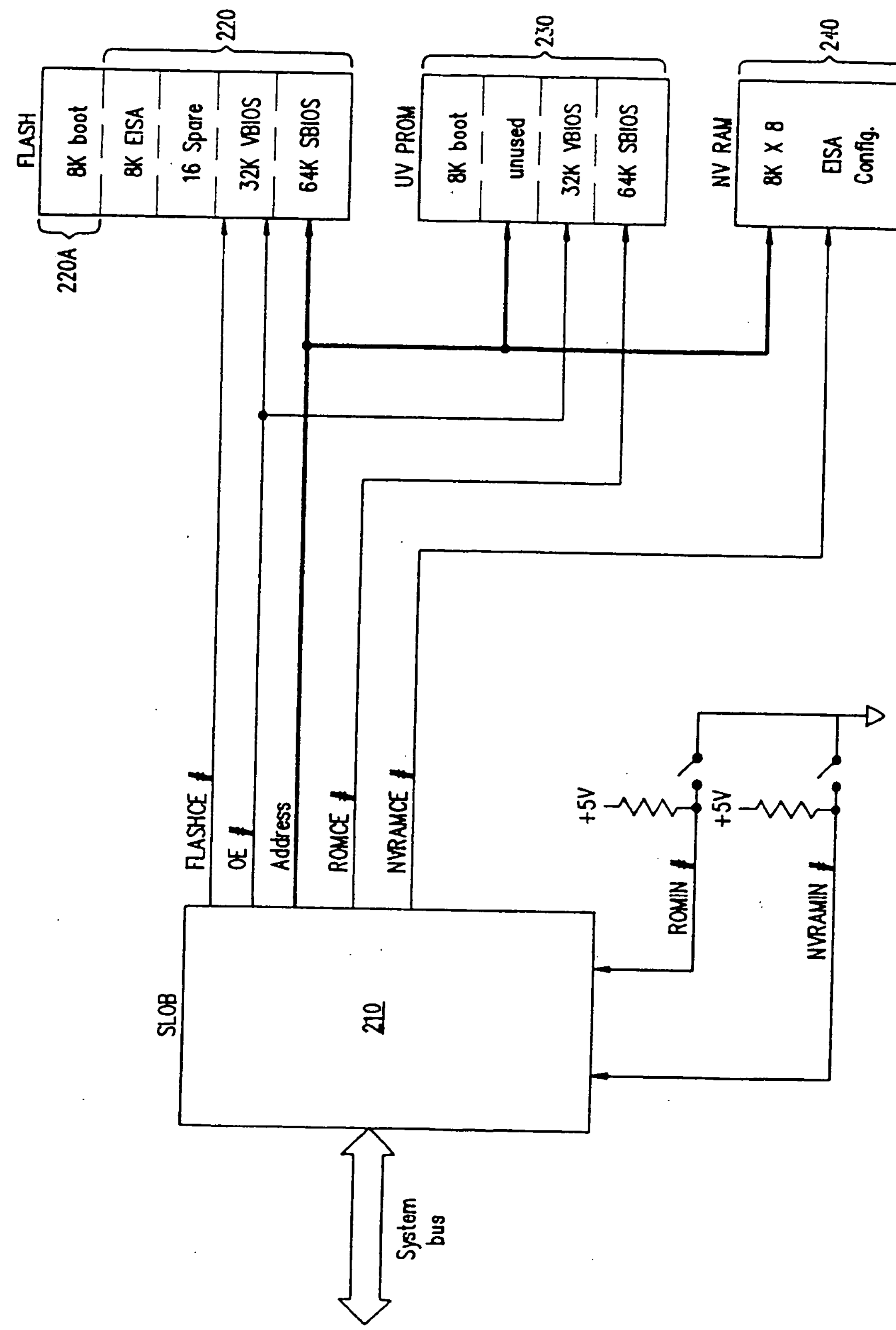
FLASH
8K boot
8K EISA
16 Spare
32K VBIOS
64K SBIOS
220
220A
UV PROM
8K boot
unused
32K VBIOS
64K SBIOS
230
NV RAM
8K X 8
EISA Config.
240
FLASHCE #
OE #
Address
ROMCE #
NVRAMCE #
+5V
+5V
ROMIN #
NVRAMIN #
SLOB
210
System bus

Figure 2

THIS PAGE BLANK (USPTO)

THIS PAGE BLANK (USPTO)